Image Processing and Effects with Core Image

Session 320

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

At the convention center, a movie will be placed here

Image Processing and Effects with Core Image

David Hayward
Pixel Perfectionist

What We Will Discuss Today

- A quick introduction to Core Image
  - Basic architecture and optimization approach
- Getting started with Core Image
  - Five simple steps to process an image
- Writing CIFilters
  - Several great examples
- Final thoughts
  - Debugging tips and other resources

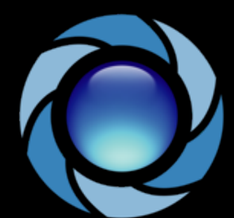
A Quick Introduction to Core Image

Meet Core Image

Meet Core Image

- Core Image is an ultra-fast image processing framework
  - Great effects for fun or profit
  - Works on real-time video with Core Video
  - Works in user interface with Core Animation
  - Runs optimized on all hardware
  - Floating point accuracy, full color-managed pipeline

- Core Image has a base set of over 125 filters
  - Custom filters packed as Image Units can extend the base set



Core Image in Use

Meet Core Image

- Filter kernels are written architecture-independent
  - Language is C-like
  - Has vector types
  - Is a subset of OpenGL shading language
- Core Image can execute filters on GPU or multicore CPU as desired
- Leverages OpenGL and/or OpenCL

Basic Concept

Filters perform per pixel operations on an image

Original

Sepia Filter

Sepia Tone applied

The final result is a new image

Basic Concept

Filters can be chained together

Original

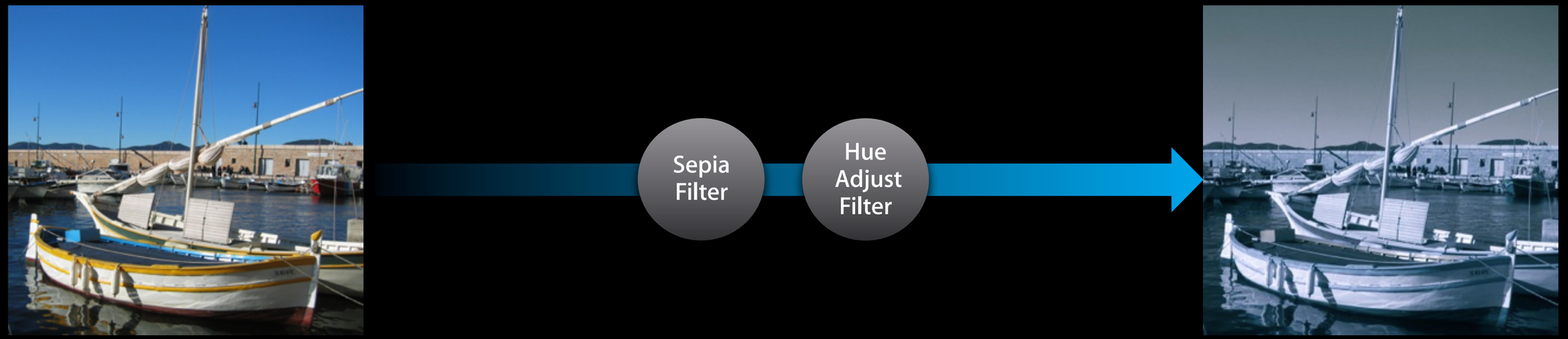
Sepia Tone applied

Sepia Tone + Hue Adjust

This allows for complex effects

Basic Concept

Filters can be concatenated



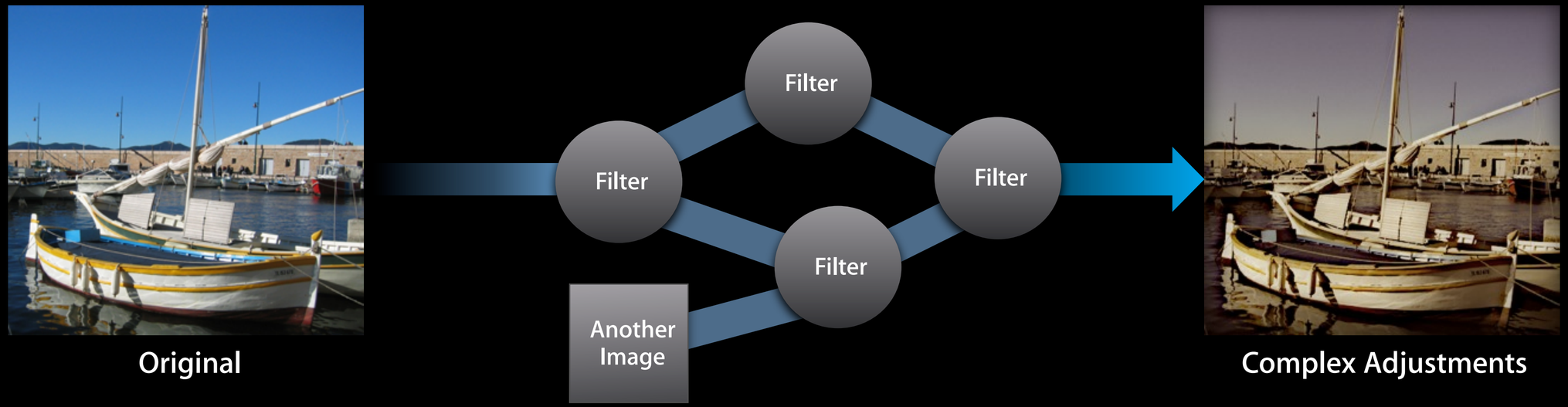
Original

Sepia Tone + Hue Adjust

This reduces intermediate buffers

Basic Concept

Complex filter trees are supported



Core Image will optimize

Core Image Filter Variety

- 4 Geometry adjustme
- 12 Distortion effects
- 7 Blurs
- 2 Sharpens
- 6 Color adjustments
- 10 Color effects
- 15 Stylize
- 5 Halftone effects
- 15 Tile effects
- 7 Generators
- 9 Transitions
- 22 Composite operations
- 8 Reduction operations and more

histogram needed for last image

David - were you going to provide this image for us?

Core Image Optimizes Your Rendering Tree

- Runtime contains a just-in-time optimizing compiler
  - Optimization is deferred until draw time
  - Evaluates just-what's-needed to display
- Filter trees are pruned and concatenated for optimal performance
- Code for entire tree is globally optimized
  - Traditional compiler optimizations (e.g., CSE and peephole)
  - Image processing domain-specific optimizations

Core Image Domain-Specific Optimizations

- Performs optimizations that general compilers can't
  - Concatenates sequential matrix operations
  - Concatenates sequential alpha-premul/alpha-unpremul
  - Reorders scale operations if possible
  - Only does color management when needed
  - Tiles large images
- Optimization improves performance and precision

- 20% faster in Snow Leopard

Getting Started with Core Image

Images, Filters, and Contexts, Oh My!

- CIImage
  - Immutable object that represents either
    - A pixelated image from memory or disk
    - The result of a filter
- CIFilter
  - A mutable object that represents a recipe to make a CIImage
  - Typically has key/value input parameters to affect output
- CIContext
  - The destination where Core Image should draw results
  - Can be based on OpenGL or Core Graphics contexts

A Simple Five-Step Program

1. Create a CIImage object
2. Create a CIFilter object
3. Set filter parameters
4. Create a CIContext object
5. Draw the filter output image into the context

Create Your CIImage

1

- From a CGImageRef

```
image = [CIImage imageWithCGImage:cgimg];
```

- From a URL pointing to a image file

```
image = [CIImage imageWithContentsOfURL:url];
```

- From data

```
image = [CIImage imageWithData:bytesPerRow:size:format:colorSpace:];
```

Create a CIFilter

- CIFilters are created by name

```
filter = [CIFilter filterWithName: @"CISepiaTone"];

filter = [CIFilter filterWithName: @"CIHueAdjust"];

filter = [CIFilter filterWithName: @"CIRippleTransition"];
```

- Custom filters are created by name too…

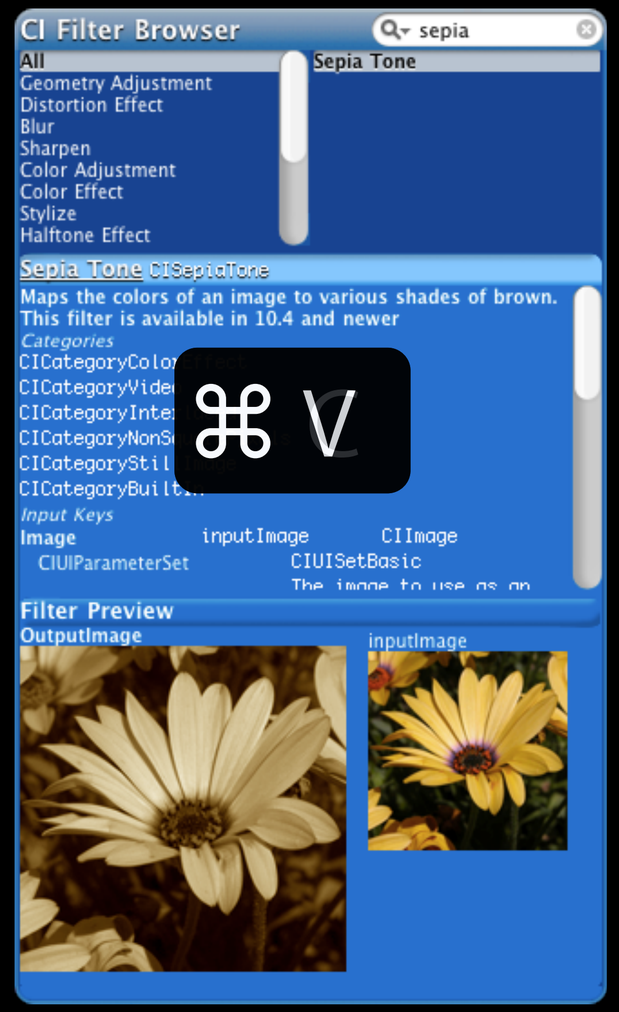
```
filter = [CIFilter filterWithName: @"myCustomFilter"];
```

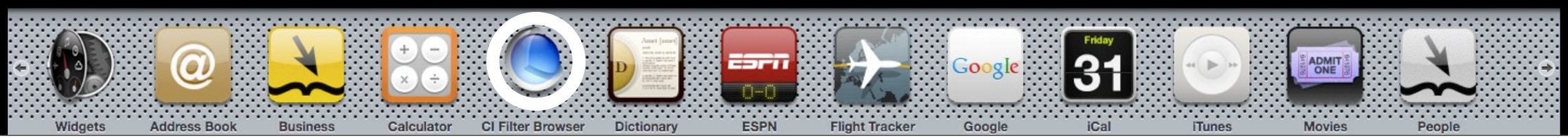
Create a CIFilter

2

- Use the CI Filter Browser dashboard widget
  - Installed with the developer tools
  - Browse and search for filters
  - Get all the details
  - Copy code template

```
[CIFilter filterWithName:@"CISepiaTone"
      keysAndValues:
      kCIInputImageKey, <#CIImage#>,
      kCIInputIntensityKey, <#NSNumber#>,
      nil];
```





Set Filter Parameters

3

- Filter parameters are set through key-value coding
- The input CIImage passed as a filter parameter

```
[filter setValue:myCIImage forKey:kCIInputImageKey];
```

- Setting a filter input with a value

```
[filter setValue:[NSNumber numberWithFloat:0.8f]
          forKey:kCIInputIntensityKey];
```

Create Your CIContext

- From a CGContext

```
context = [CIContext contextWithCGContext:cgcontext options:nil];
```

- From an OpenGL context

```
context = [CIContext contextWithCGLContext:glcontext options:nil];
```

- NSView provides a convenience wrapper

```
context = [[NSGraphicsContext currentContext] CIContext];
```

Create Your CIContext

4

- To get the best performance using a CG-based CIContext
  - Opt-in to QuartzGL acceleration by adding QuartzGLEnable to Info.plist
  - Allows CI rendering to be fully GPU accelerated
  - Avoids read-back from the graphics card to CG's backing store

Drawing the Result

- Get the filter result through key-value coding

```
resultImage = [filter valueForKey: kCIOutputImageKey];
```

- Tell your CIContext to draw the image

```
[context drawImage: resultImage
           atPoint: CGPointZero
          fromRect: [resultImage extent]];
```

- Also, to get the results for

```
[context createCGImage:];

[context render:toBitmap:rowbytes:bounds:format:colorspace];
```

All Together Now

Just a few lines of Core Image code…

```
CIImage* image = [CIImage imageWithContentsOfURL: myURL];

image = [[[CIFilter filterWithName:@"CISepiaTone" keysAndValues:
          kCIInputImageKey, image,
          kCIInputIntensityKey, [NSNumber numberWithFloat:1.0], nil]
    valueForKey: kCIOutputImageKey];

image = [[[CIFilter filterWithName:@"CIHueAdjust" keysAndValues:
          kCIInputImageKey, image,
          kCIInputAngleKey, [NSNumber numberWithFloat:1.57], nil]
    valueForKey: kCIOutputImageKey];

[context drawImage: image atPoint:CGPointZero fromRect:[image extent]];
```

All Together Now

…Core Image will do all this OpenGL work

```
CGLQueryRendererInfo(2, 0x00000000, 0);
CGLDescribeRenderer(0x00246210, 0, kCGLRPRendererID);
CGLChoosePixelFormat({kCGLPFAColorSize, 32, kCGLPFANoRecovery, kCGLPFAAccelerated, 96, kCGLPFARendererID, 16918024}, 0x13c398a0, 1);
CGLCreateContext(0x13c398a0, 0x00000000, 0x00825800);
CGLSetSurface(0x00825800, {20, 42, 1024, 1024}, {0, 0, 1024, 1024});
glScissor(0, 0, 1024, 1024);
glViewport(0, 0, 1024, 1024);
glViewport(0, 0, 1024, 1024);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 1024, 0, 1024, -1, 1);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glClearColor(0, 0, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
glGetString(GL_RENDERER);
glGetString(GL_VERSION);
CGLGetVirtualScreen(0x00825800);
CGLGetVirtualScreen(0x00825800);
CGLQueryRendererInfo(2, 0x000cb820, -1730100884);
CGLDescribeRenderer(0x13c2cae0, 0, kCGLRPAccelerated);
CGLDescribeRenderer(0x13c2cae0, 0, kCGLRPVideoMemory);
CGLDescribeRenderer(0x13c2cae0, 1, kCGLRPAccelerated);
CGLDestroyRendererInfo(0x13c2cae0);
glGetFloatv(GL_MAX_VIEWPORT_DIMS, 0xbffff148);
glGetFloatv(GL_MAX_RECTANGLE_TEXTURE_SIZE_EXT, 0xbffff184);
glGetIntegerv(GL_MAX_3D_TEXTURE_SIZE, 0x00246b40);
glGetString(GL_VERSION);
glGetString(GL_EXTENSIONS);
glGetString(GL_RENDERER);
glGetProgramivARB(GL_VERTEX_PROGRAM_ARB, GL_MAX_PROGRAM_NATIVE_PARAMETERS_ARB, 0xbffff178);
glGetProgramivARB(GL_VERTEX_PROGRAM_ARB, GL_MAX_PROGRAM_NATIVE_TEMPORARIES_ARB, 0xbffff170);
glGetProgramivARB(GL_VERTEX_PROGRAM_ARB, GL_MAX_PROGRAM_NATIVE_ATTRIBS_ARB, 0xbffff174);
glGetProgramivARB(GL_VERTEX_PROGRAM_ARB, GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB, 0xbffff16c);
glGetProgramivARB(GL_VERTEX_PROGRAM_ARB, GL_MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB, 0xbffff168);
```

All Together Now

…Core Image will do all this OpenCL work

```
aa = clCreateContextFromType( NULL,  CL_DEVICE_TYPE_CPU, NULL, NULL, 0);
clRetainContext(aa);
clGetContextInfo(aa, CL_CONTEXT_DEVICES, 0, NULL, 4);
clGetContextInfo(aa, CL_CONTEXT_DEVICES, 4, { bb }, 4);
clGetDeviceInfo(bb, CL_DEVICE_TYPE, 8, 2, 8);
clGetDeviceInfo(bb, CL_DEVICE_MAX_COMPUTE_UNITS, 4, 4, 4);
cc = clCreateCommandQueue(aa, bb, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE | CL_QUEUE_PROFILING_ENABLE, 0)
clGetDeviceInfo(bb, CL_DEVICE_MAX_SAMPLERS, 4, 16, 0);
clGetDeviceInfo(bb, CL_DEVICE_MAX_PARAMETER_SIZE, 4, 4096, 0);
clCreateSampler(aa, CL_FALSE, CL_ADDRESS_CLAMP, CL_FILTER_LINEAR, 0);
clCreateSampler(aa, CL_FALSE, CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_LINEAR, 0);
clCreateSampler(aa, CL_FALSE, CL_ADDRESS_REPEAT, CL_FILTER_LINEAR, 0);
gg = clCreateSampler(aa, CL_FALSE, CL_ADDRESS_CLAMP, CL_FILTER_NEAREST, 0);
clCreateSampler(aa, CL_FALSE, CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_NEAREST, 0);
clCreateSampler(aa, CL_FALSE, CL_ADDRESS_REPEAT, CL_FILTER_NEAREST, 0);
clCreateSampler(aa, CL_TRUE, CL_ADDRESS_CLAMP, CL_FILTER_LINEAR, 0);
clCreateSampler(aa, CL_TRUE, CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_LINEAR, 0);
clCreateSampler(aa, CL_TRUE, CL_ADDRESS_REPEAT, CL_FILTER_LINEAR, 0);
clCreateSampler(aa, CL_TRUE, CL_ADDRESS_CLAMP, CL_FILTER_NEAREST, 0);
clCreateSampler(aa, CL_TRUE, CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_NEAREST, 0);
clCreateSampler(aa, CL_TRUE, CL_ADDRESS_REPEAT, CL_FILTER_NEAREST, 0);
clRetainContext(aa);
clRetainCommandQueue(cc);
hh = clCreateImage2D(aa,  CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR ,  { CL_ARGB , CL_UNORM_INT8 }, 1024, 1024, 4096, pelon, 0);
clRetainMemObject(hh);
clGetImageInfo(hh, CL_IMAGE_ROW_PITCH, 4, <param_value>, 0);
clGetImageInfo(hh, CL_IMAGE_WIDTH, 4, <param_value>, 0);
clGetImageInfo(hh, CL_IMAGE_HEIGHT, 4, <param_value>, 0);
clGetImageInfo(hh, CL_IMAGE_DEPTH, 4, <param_value>, 0);
clGetImageInfo(hh, CL_IMAGE_FORMAT, 8, <param_value>, 0);
clReleaseMemObject(hh);
clGetDeviceInfo(bb, CL_DEVICE_IMAGE2D_MAX_WIDTH, 4, 8192, 0);
clGetDeviceInfo(bb, CL_DEVICE_IMAGE2D_MAX_HEIGHT, 4, 8192, 0);
clGetDeviceInfo(bb, CL_DEVICE_MAX_MEM_ALLOC_SIZE, 8, 1073741824, 0);
clGetDeviceInfo(bb, CL_DEVICE_MAX_SAMPLERS, 4, 16, 0);
```

Demo

Sample Core Image application

And Now to the Main Attraction:

Writing CIFilters

Frank Doepke
Pixels on Two Wheels

Writing a CIFilter

- Why would you do it
  - Filter does not exist in the existing set
  - A compound problem can be better described in a custom filter than chaining existing filters
  - Did I mention that you can sell them?

Writing a CIFilter

- What is involved
  - Write one or more filter kernels in the CI kernel language
  - Wrap it in Obj-C to create a filter or Image Unit
- Is there any information available
  - Image Unit Tutorial and CI reference on ADC site
- What tools do we have
  - Quartz Composer for rapid prototyping
  - XCode for developing Image Units

The CI Filter Anatomy

- The low-level code in the CIKernel
  - Written in CIKernel language
  - Here is where the imaging algorithm lives
  - Produces exactly one output pixel
  - Can read from multiple input pixels

The CI Filter Anatomy

- The high-level code in Obj-C
  - Set up samplers to access images
  - Load the CIKernel objects
  - Set up all the parameters for the kernel
  - Define the Domain of Definition (DOD) of output image
  - Define the Region of Interest (ROI) of input images

The CI Filter Anatomy

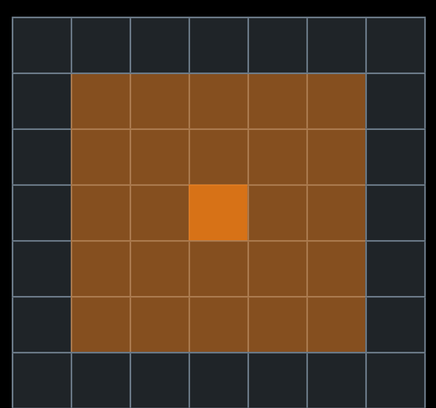
Kernel gets asked to produce output pixel at X,Y

```
kernel vec4 negative (sampler src)
{
    vec4 s;

    s = sample (src, samplerCoord (src));
    s.rgb = s.aaa - s.rgb;

    return s;
}
```

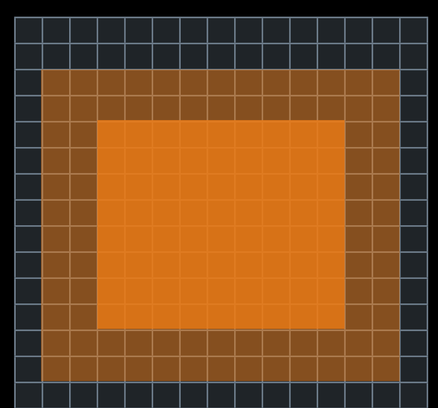
The imaging algorithm

DOD and ROI Concept

- The Domain of Definition—"What the filter will render"
  - Defines the rectangle the filter produces
  - Pixels outside the DOD must be transparent black
- The Region of Interest—"What the filter needs to render"
  - The rectangle enclosing all pixels of an input sampler needed to produce the given output rectangle
    - Example: A kernel averages the pixels in a two-pixel radius



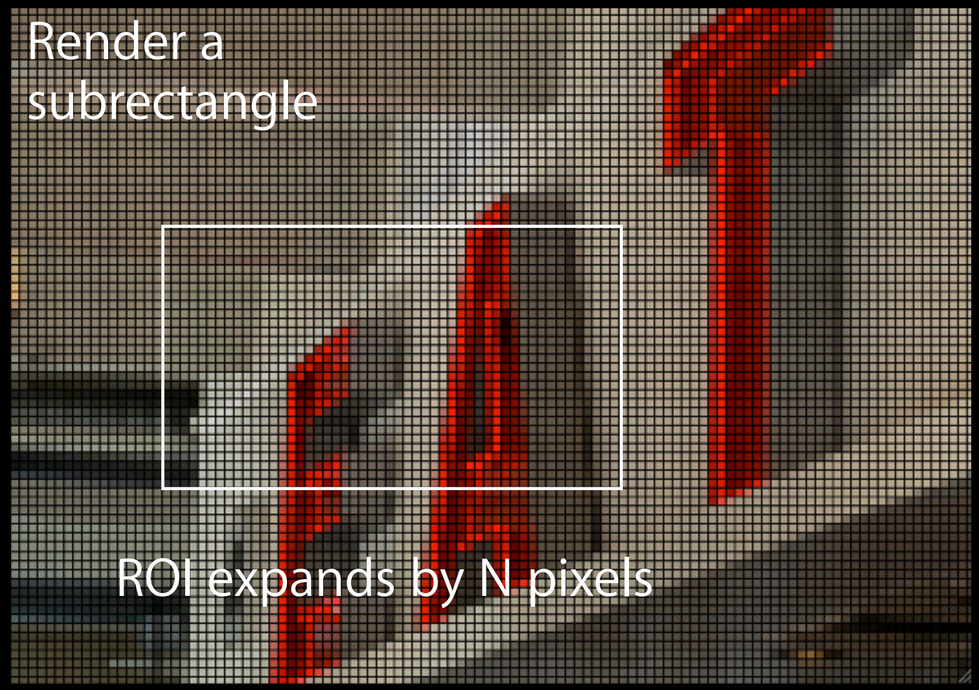
Sample two pixels around the current pixel



Inset the destination rectangle by two pixels to the outside

DOD and ROI Concept

- Why is the ROI so important?
  - Core Image will tile the images for optimized rendering
  - Without knowing what a filter needs the tiles might not include all the pixels the kernel wants to sample



Let’s Start Writing a Filter

- Pixellate example from Image Unit Tutorial
  - Create a block of pixels defined by a radius with the color of the center pixel and modifies the opacity to create large round dots

Let's Start Writing a Filter

```
kernel vec4 roundPixellate(sampler src, float radius)
{
    vec2    positionOfDestPixel, centerPoint;
    float   d;
    vec4    outValue;
    float   smooth = 0.5;

    positionOfDestPixel = destCoord();
    centerPoint = positionOfDestPixel;
    centerPoint.x = centerPoint.x + radius
                  - mod(positionOfDestPixel.x, radius * 2.0);
    centerPoint.y = centerPoint.y + radius
                  - mod(positionOfDestPixel.y, radius * 2.0);
    d = distance(centerPoint, positionOfDestPixel);

    outValue = sample(src, samplerTransform(src, centerPoint));
    outValue = outValue * smoothstep((d - smooth), d, radius);

    return outValue;
}
```

Input Image and radius of our dots.

Declare local variables. We use two vec2 data type variables for the center of our block and the destination pixel position.

Get the position, in working-space coordinates, of the pixel currently being computed.

Calculate the coordinate of the destination pixel.

Distance from the center of our dot.

Fetch the pixel at the center.
Create an antialiased edge using smoothstep.

Return the premultiplied output pixel.

Let's Talk About Some Language Features

`sampler` Requires a CISampler object when applied. The sampler does all the transformation work to access the image.

`__table` A qualifier for a sampler type.

`float, vec2, vec3, vec4` Requires an NSNumber or CIVector.

`__color` A color that will be matched to the CIContext working color space when passed into the program. It requires a CIColor object when applied. To the kernel program it appears to be a vec4 type in premultiplied RGBA format.

Let’s Talk About Some Language Features

`destCoord` Returns the position, in working space coordinates, of the pixel currently being computed.

`samplerCoord` Returns the position, in sampler space, of the sampler src that is associated with the current output pixel.

`sample` Returns the pixel value produced from sampler src at the position point, where point is specified in sampler space.

`unpremultiply/premultiply` Convert RGBA from and into premultiplied RGBA which is the expected image format in Core Image.

Let's Talk About Some Language Features

`smoothstep` From GLSL that allows you to interpolate between 0 and 1 in a given range.

`sin/cossin/tan` Trigonometric functions.

`compare` For each component, returns x < 0 ? y : z.

`? :` Operator for conditional expressions.

"for" loops and "if" statements are not supported as kernels do not support jumps. Loops must be unrolled. This allows for vectorized code.

The High-Level Pieces

- ROI function

```
- (CGRect)regionOf: (int)sampler  destRect: (CGRect)rect  userInfo: (NSNumber *)radius
{
    return CGRectInset(rect, -[radius floatValue], -[radius floatValue]);
}
```

- Attributes

```
- (NSDictionary *)customAttributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:
       [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  1.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:  200.00], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  4.00], kCIAttributeDefault,
            [NSNumber numberWithDouble:  1.00], kCIAttributeIdentity,
            kCIAttributeTypeScalar,          kCIAttributeType,
            nil],                              @"inputScale",
        nil];
}
```

The High-Level Pieces

- The outputImage method

```
- (CIImage *)outputImage
{
    CISampler *src;

    // Set up the sampler to use the input image
    src = [CISampler samplerWithImage:inputImage];

    // Set the region-of-interest method for the kernel routine
    [_PixellateFilterKernel setROISelector:@selector(regionOf:destRect:userInfo:)];
    // Apply the filter to the kernel, passing the sampler and scale
    return [self apply:_PixellateFilterKernel, src, inputRadius,
    // This option specifies the domain of definition of the destination image
        kCIApplyOptionDefinition, [[src definition] insetByX:-[inputRadius floatValue]
                                                           Y:-[inputRadius floatValue]],
    // This option sets up the ROI method to gets radius value
        kCIApplyOptionUserInfo, [NSNumber numberWithFloat:radius], nil];
}
```

Pitfalls and Other Oopsies

- Parameters passed to a kernel have to be objects, e.g., 0.5f has to be passed in as [NSNumber numberWithFloat:0.5f]
- Colors have to be CIColors
- If you need to pass data to a kernel load it into an image and make sure that
  - The image is created with the colorspace set to nil to avoid colormatching
  - The kernel is the sampler is tagged with __table sampler

Filter creators must supply Core Image with color components that are premultiplied by the alpha value

Taking Pixellate Up a Notch

- Giving the dots a three-dimensional appearance with some shine

- Now we have two samplers
- What happens to our ROI

Let’s Do Some Damage

Build Image Unit using Xcode and demonstrate using Pixelmator

Stacking Filters vs. Custom Kernel

- Chroma screen
- First, with the given CIFilters
  - Create a color cube image
  - In that cube, change the “chroma screen” color to transparent black
  - Use the CIColorCube filter
- Second, write a kernel that takes a color value and masks with “fuzziness” parameter
  - Convert the color to YCbCr and compare the Cb Cr components within a certain tolerance

Stacking Filters vs. Custom Kernel

```
kernel vec4 chromaScreen(sampler image, sampler backgroundImage, __color color,
                                                float threshhold, float smoothing)
{
   float    Y, Cb, Cr;
   float    maskY, maskCb, maskCr;
   vec4     sourcePixel = sample(image, samplerCoord(image));
   vec4     backgroundPixel = sample(backgroundImage, samplerCoord(backgroundImage));
   float    blendValue;

   maskY = 0.2989 * color.r + 0.5866 * color.g + 0.1145 * color.b;
   maskCr = 0.7132 * (color.r - maskY);
   maskCb = 0.5647 * (color.b - maskY);

   Y = 0.2989 * sourcePixel.r + 0.5866 * sourcePixel.g + 0.1145 * sourcePixel.b;
   Cr = 0.7132 * (sourcePixel.r - Y);
   Cb = 0.5647 * (sourcePixel.b - Y);

   blendValue = smoothstep(threshhold, threshhold + smoothing ,
                                          abs((Cr - maskCr) + (Cb - maskCb)));
   sourcePixel = sourcePixel * blendValue;

   return sourcePixel + (backgroundPixel * (1.0 - blendValue));
}
```

Demo

Coding it up in Quartz Composer

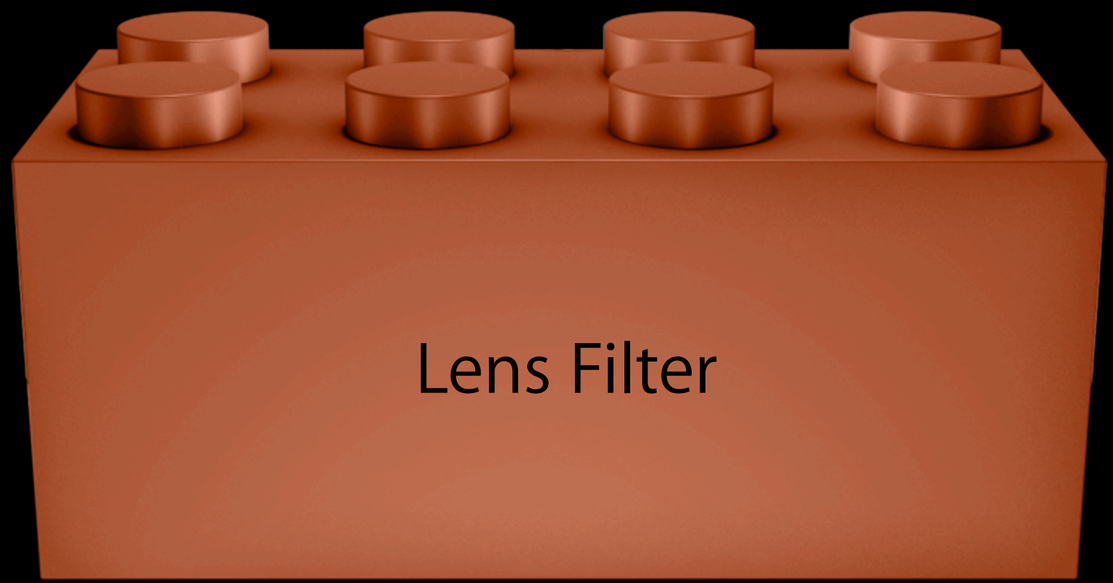
Talking About More Complex Problems

- The salami strategy for complex problems
  - Slice the algorithm—you don’t have to eat it as a whole
  - A filter can use more than one kernel to produce the outputImage
  - A filter can call other CIFilters, for instance, to composite the result of two kernels

Talking About More Complex Problems

Lens Filter example from Image Unit Tutorial

- One kernel to produce the lens scale and shine
- Second kernel to produce the programatic ring image
- Both resulting images get combined using CISourceOverCompositing

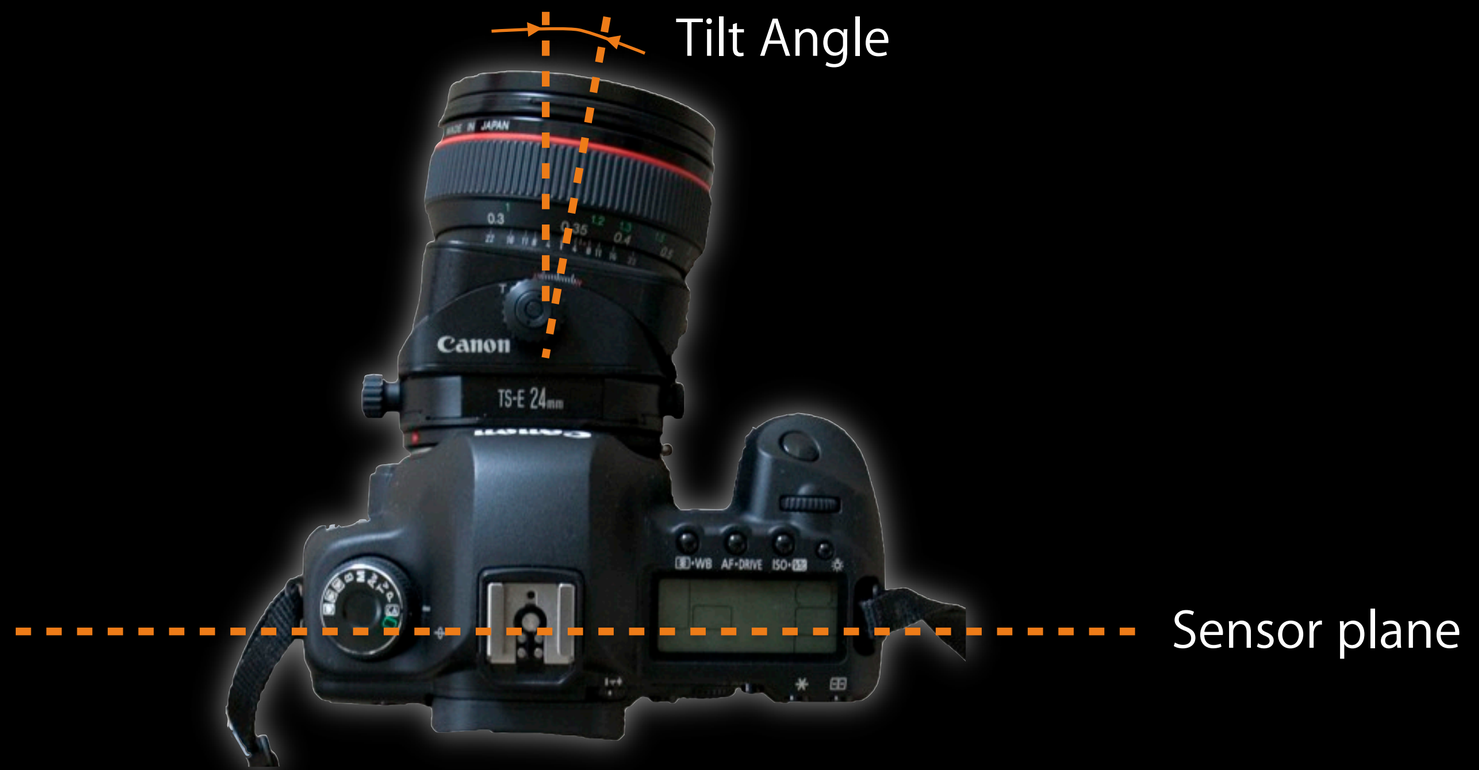


A ‘Mini’ Example All in One App

Alexandre Naaman
That’s How He Rolls

Miniaturization Sample

- Goal: Demonstrate how to simulate effect of a tilt and shift lens
- Use one standard Core Image filter and one custom filter



Standard Lens vs. Tilt and Shift Lens

Blurry

Sharp

Blurry

“Fake Miniatures”: Image Components

- First pass at this algorithm: Use base image and blurred image
- outputImage = (t * baseImage) + ((1 - t) * blurredImage)
  - t is a function of the distance from a user defined line in the image
- In CI kernel language:

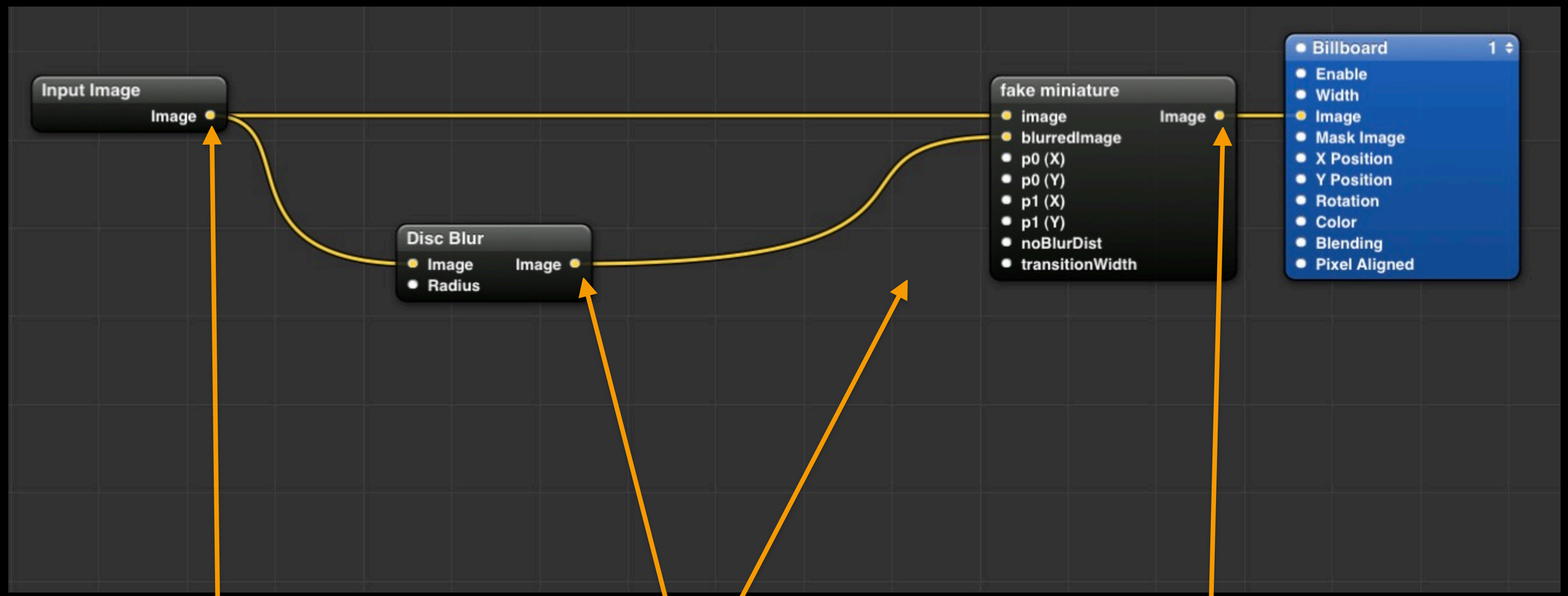
```
output = mix ( baseImagePixel, blurredImagePixel, t );
```

How Well Does It Work?

Actual photo from T&S lens

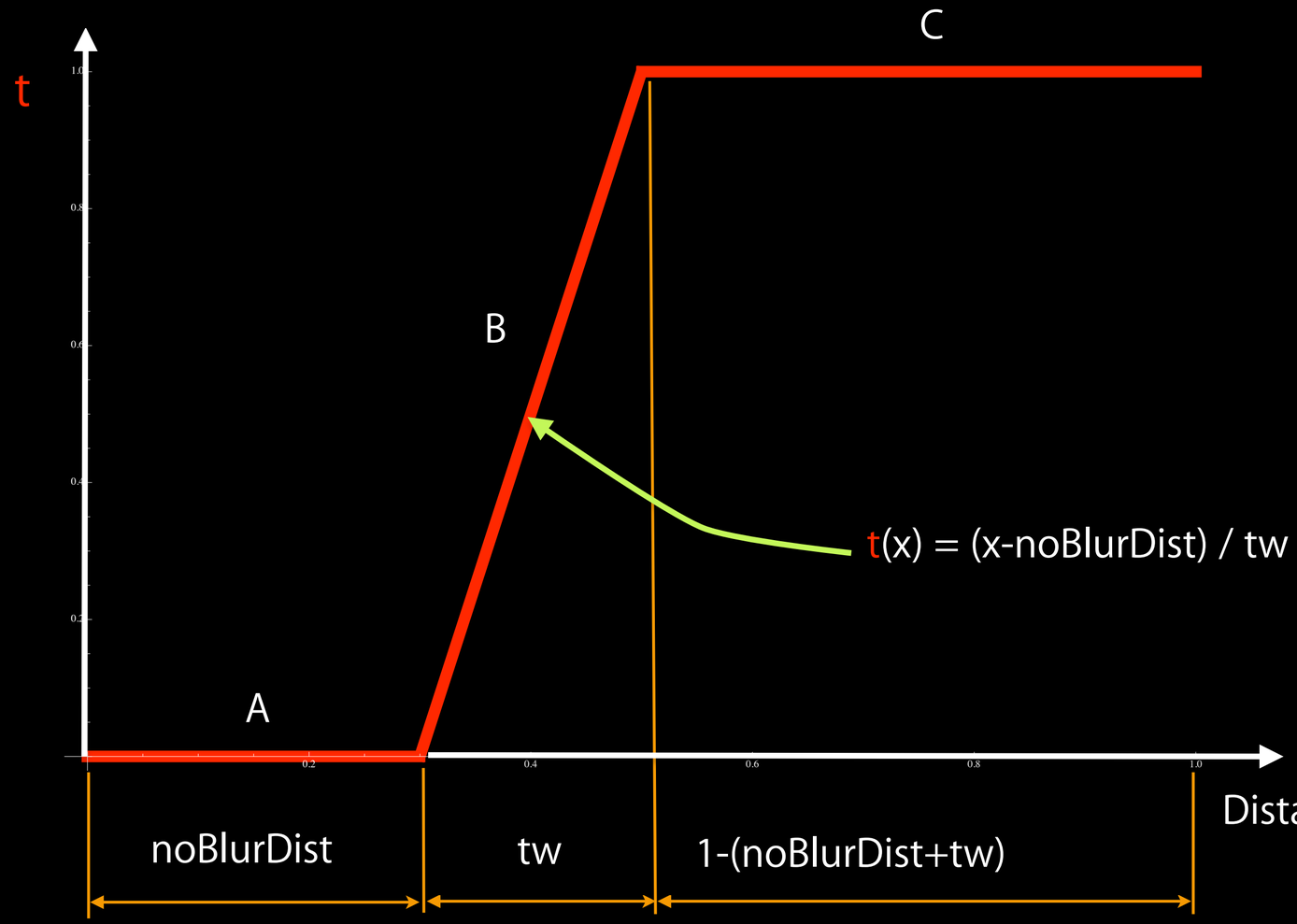
Using Core Image Filter and regular lens

Simplified Quartz Composer Composition

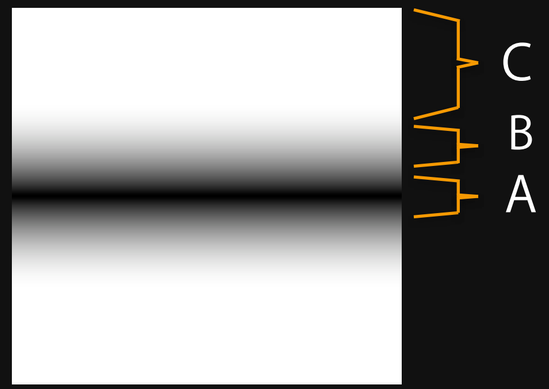


Our Blending Function: Distance from Line

- Really just a line equation clamped at 0.0 and 1.0



A: Original image only
B: Transition width, use both images
C: Blurred image only



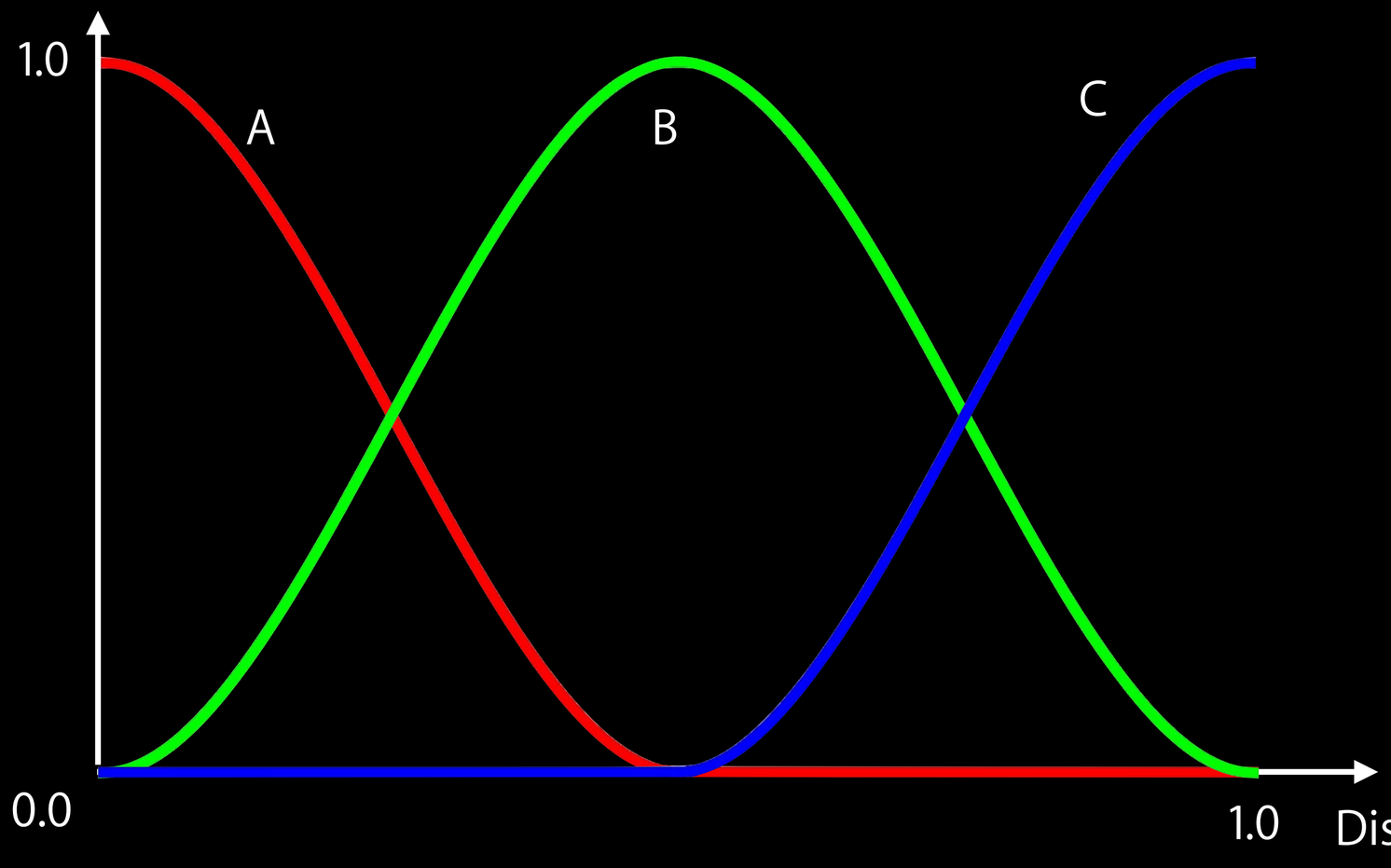
Fake Miniature Kernel Code

```
float distFromPointToLine(vec2 p0,vec2 p1,vec2 p)
{
    float denom = (p1.x-p0.x)*(p1.x-p0.x) + (p1.y-p0.y)*(p1.y-p0.y);
    float s     = ((p0.y-p.y)*(p1.x-p0.x)-(p0.x-p.x)*(p1.y-p0.y)) / denom;

    return abs(s)*sqrt(denom);
}
```

```
kernel vec4 fakeMiniature(sampler image,sampler blurredImage,vec2 p0,vec2 p1,
                          float noBlurDist,float transitionWidth)
{
    vec2 pt    = samplerCoord(image);
    vec4 col0  = sample(image, pt);
    vec4 col1  = sample(blurredImage, samplerCoord(blurredImage));
    vec2 pNorm = pt / samplerSize(image);

    float dist = distFromPointToLine ( p0, p1, pNorm );

    float t = clamp( (dist - noBlurDist) / transitionWidth, 0.0, 1.0);

    col0.rgb = mix(col0.rgb, col1.rgb, t);

    return col0; // retains alpha value from original input image
}
```

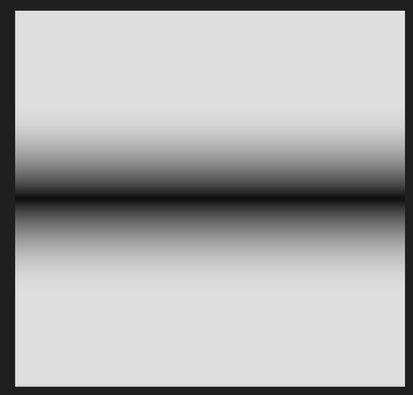
Fancier Method: Multiple Blur Radii

- Must sum 1.0 for all values of x



A: Original input image
B: Medium radius blur image
C: Large radius blur image

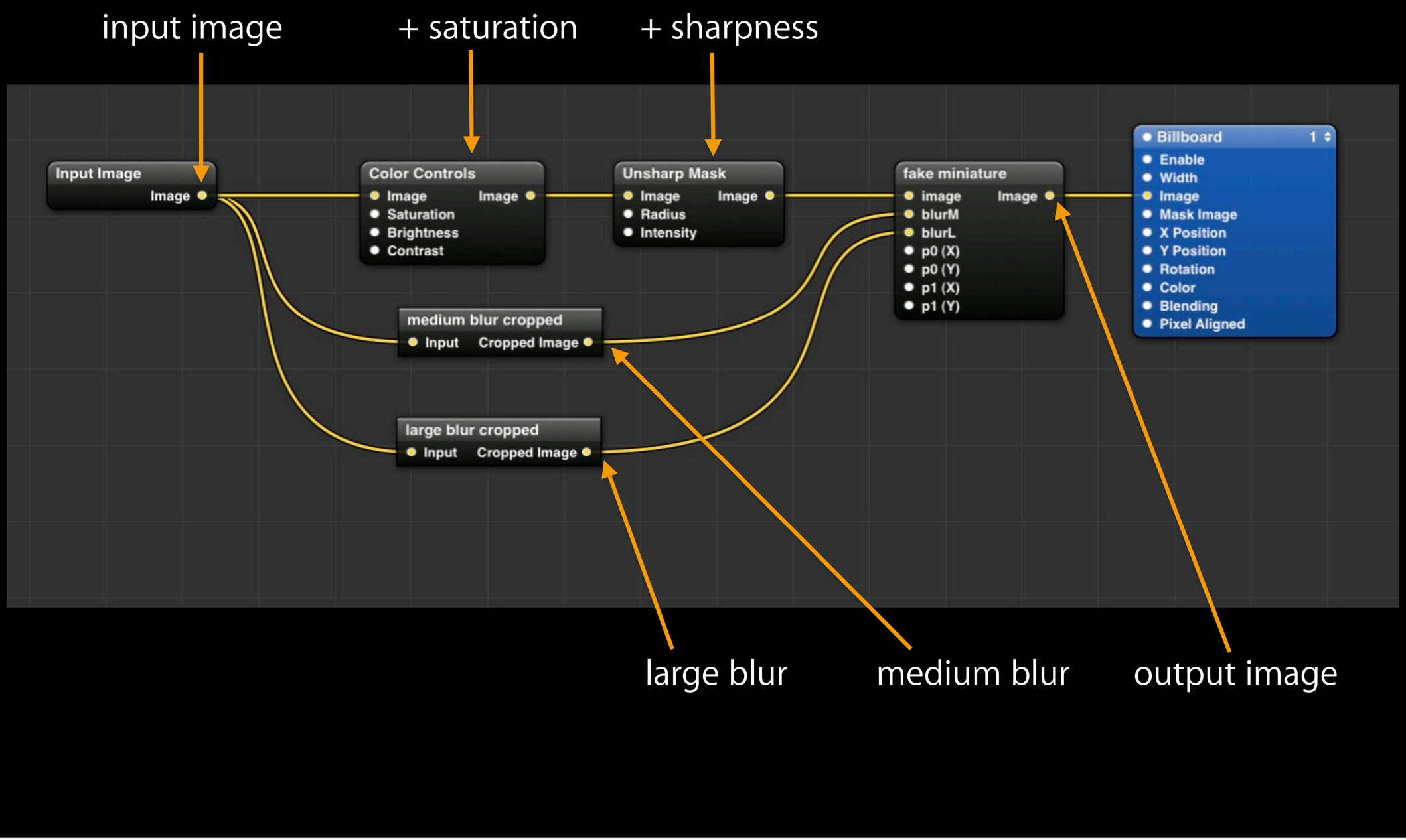
B + C =



Fancier Technique: Kernel Code

```
kernel vec4 fakeMiniature(sampler image,sampler blurMed,sampler blurLrg,vec2 p0,vec2 p1)
{
   vec2 pt     = samplerCoord(image);
   vec4 col0   = sample(image, pt);
   vec4 colMed = sample(blurMed, samplerCoord(blurMed));
   vec4 colLrg = sample(blurLrg, samplerCoord(blurLrg));
   vec2 pNorm  = pt / samplerSize(image);

   float dist = distFromPointToLine ( p0, p1, pNorm ) * 2.0;

   float a = smoothstep (0.0, 1.0,  1.0 - 2.0 * dist);
   float c = smoothstep (0.0, 1.0, -1.0 + 2.0 * dist);
   float b = 1.0 - (a + c);

   col0.rgb = (a * col0.rgb) + (b * colMed.rgb) + (c * colLrg.rgb);

   return col0; // retains alpha value from original input image
}
```

More Fakeness: Saturation + Sharpness



End Result

With additional
Three-way image
Actual RGB and
combine
sharpness

A Few Caveats/Notes of Interest

- Must use CIDiscBlur, not CIGaussianBlur
- Save your money, don’t buy a tilt and shift lens
  - Buy a fast GPU instead!

Demo

Good Reads on Writing Filters…

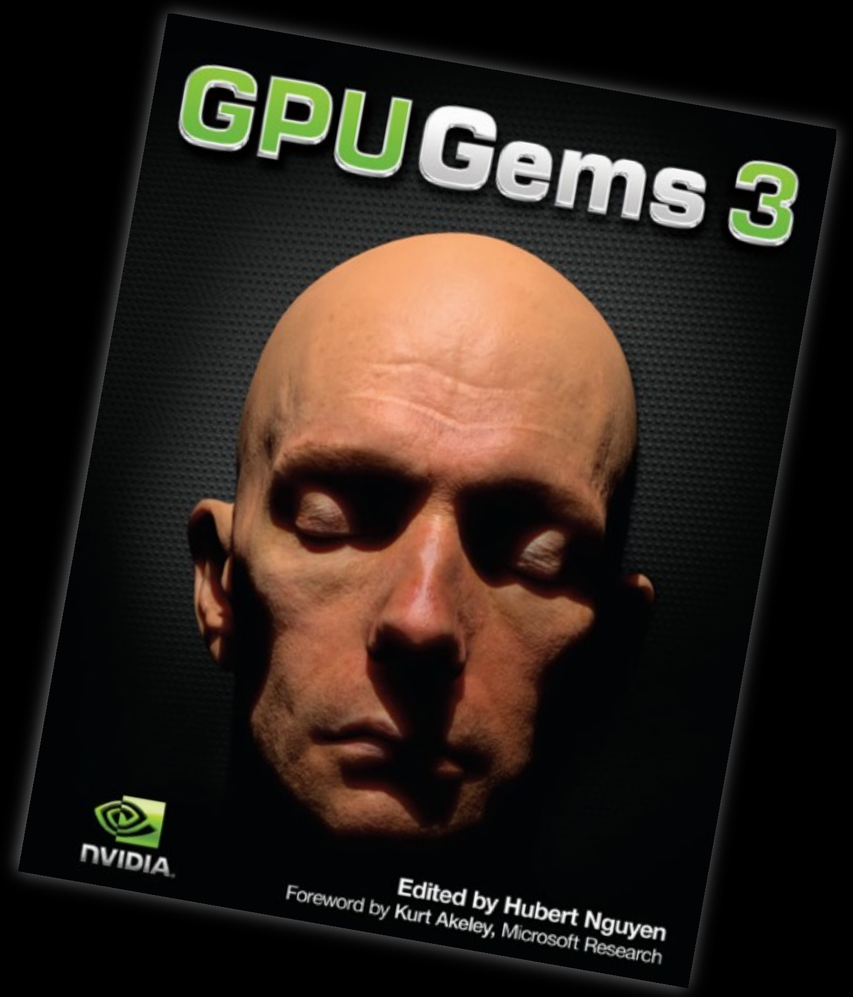
Throwing the Book at It

- To learn more about image resampling:

- "Digital Image Warping"
  - George Wolberg, 1992

Throwing the Book at It

- "GPU Gems 3"
  - Addison-Wesley, 2007
- Good Core Image sample for color object tracking



If the Image Does Not Appear as Planned…

Where to Go from Here

- QuartzDebug allows you to check your filter tree when rendering
  - Also gives performance data
- Check the Console when CI cannot execute the filter
  - ROI not tileable
  - Syntax errors in the kernel
- Make sure to test your ROI functions with large images
- Rotate and scale your input images to flush incorrect sampler transformations
- Keep the value range between kernels from –1.0 to 3.0

Where to Go from Here

- Check out the samples in /Developer/Examples/
  - Quartz/Core Image
  - Quartz Composer/Compositions/Core Image
  - Quartz Composer/Compositions/Core Image Filters
- Take advantage of iSight—GPU Gems sample on ADC: CIColorTracking
- Play with the filters—Concatenate your effects
- Not only for image editing—Great for UI and visual effects

Who to Contact

Allan Schaffer
Graphics and Imaging Evangelist
aschaffer@apple.com

Labs

Core Image	Graphics and Media Lab B Friday 10:30AM

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09